DSD - Práctica 4

[**Ejercicio 1 - Implementación de los ejemplos**](#_mqgsrccf23jr) **3**

[¿Qué es Java RMI?](#_f3p4xbz52nat) 3

[Implementación de una aplicación Java RMI](#_ebl9y9w2fbyi) 3

[Compilación](#_e1cjcgrdvvq0) 3

[Ejecución](#_mfoa9k34zyt) 4

[Simple](#_xkj8lb79nqbx) 4

[Multihebra](#_1rznshveahq2) 4

[Contador](#_grcihnmbmrpy) 5

[Ejercicio 1](#_hoz8r9zfeoh1) 7

[**Ejercicio 2 - Servidor replicado**](#_8jeqvqm23vw7) **8**

[Introducción](#_lquijayg5ubd) 8

[Desarrollo](#_973z9dkds0uj) 8

[Interfaces](#_vlud3iwznh4c) 10

[IReplicaCommunicableClient](#_qqax0jcfiaqg) 10

[IReplica](#_3bxvygnnjq6c) 11

[Implementación](#_nmwtsls9a4t) 12

[Réplica](#_gs9de9s1brir) 12

[Comunicación](#_v0jfe4yazs8u) 13

[Ejecución](#_goxhjuz352g) 15

[Extras](#_6imqixwj5mfk) 15

# Ejercicio 1 - Implementación de los ejemplos

## ¿Qué es Java RMI?

Es una API (Application Programming Interface) RMI (Remote Method Invocation) para el lenguaje de programación Java.

## Implementación de una aplicación Java RMI

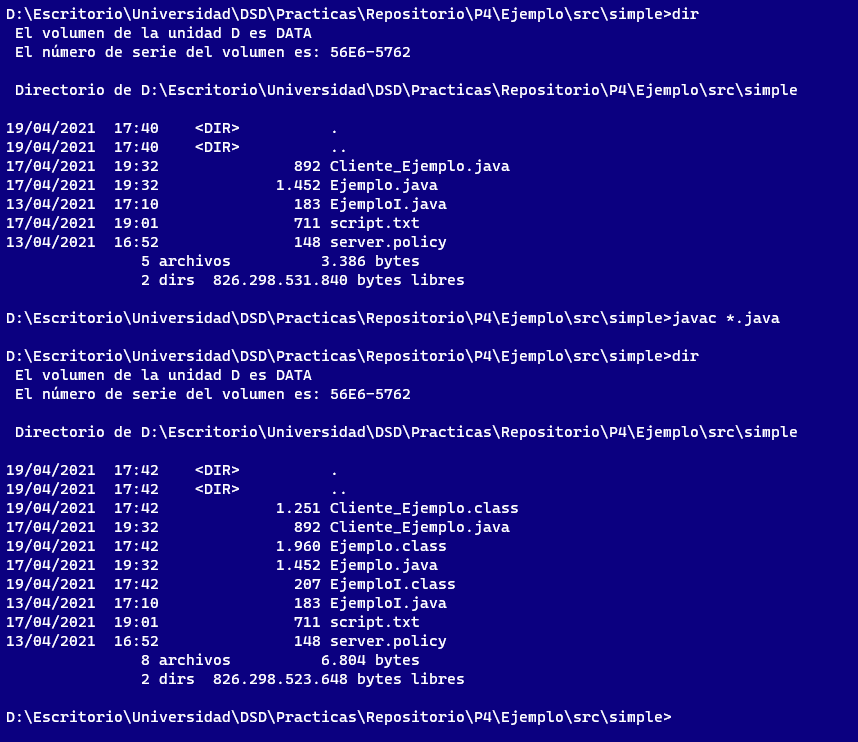
Gracias a la tecnología Java RMI, se pueden compartir objetos a través de red. Para ello, lo primero es declarar una interfaz, que describe las operaciones que puede ejecutar el objeto remoto. Posterior a ello deberemos de implementar un objeto que realice dicha interfaz.

Vamos a trabajar con una arquitectura cliente-servidor, por lo que deberemos:

* Crear el servidor
  + Instalar el gestor de seguridad
  + Crear una instancia del objeto remoto
  + Exportar el objeto remoto en el registro rmi
* Crear el cliente
  + Instalar el gestor de seguridad
  + Crear una instancia del registro rmi
  + Buscar en el registro el objeto remoto exportado por el servidor
  + Realizar las operaciones con el objeto remoto

## Compilación

> javac \*.java



Si nos fijamos, se nos han creado correctamente los archivos .class y no hemos tenido problemas en tiempo de compilación.

## 

## Ejecución

### Simple

1º Iniciar el registro

> rmiregistry



2º Ejecutar el servidor

> java -cp . -Djava.rmi.server.codebase=file:/. -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Ejemplo

Como tenemos todo en la misma carpeta, el flag -Djava.rmi.server.codebase no haría falta. Con él especificamos dónde se encuentran las definiciones de las clases del servidor.

Con esto estoy ejecutando el servidor y poniéndolo a la escucha de peticiones por parte del cliente.



3º Ejecutar el cliente

> java -cp . -Djava.security.policy=server.policy Cliente\_Ejemplo localhost 0

Cuando ejecutamos el cliente con el argumento 0 se hace una espera, debido que en el servidor en el método de escribir\_mensaje dormimos la hebra que está ejecutándose actualmente.

System.out.println("Recibida petición de proceso: " + id\_proceso);

if (id\_proceso == 0) {

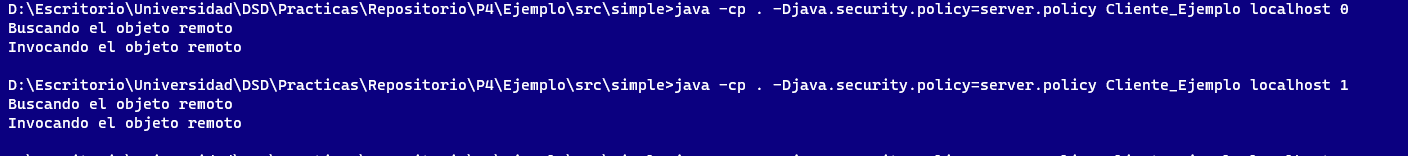
try {

System.out.println("Empezamos a dormir");

Thread.sleep(5000);

System.out.println("Terminamos de dormir");

} catch (Exception e) {



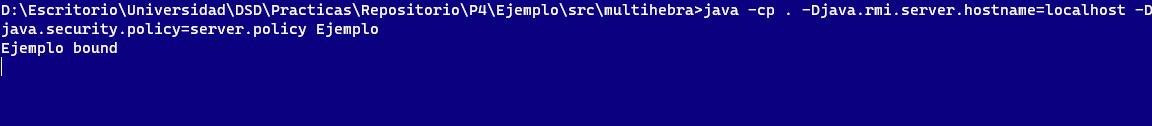
### Multihebra

1º Iniciar registro

> rmiregistry

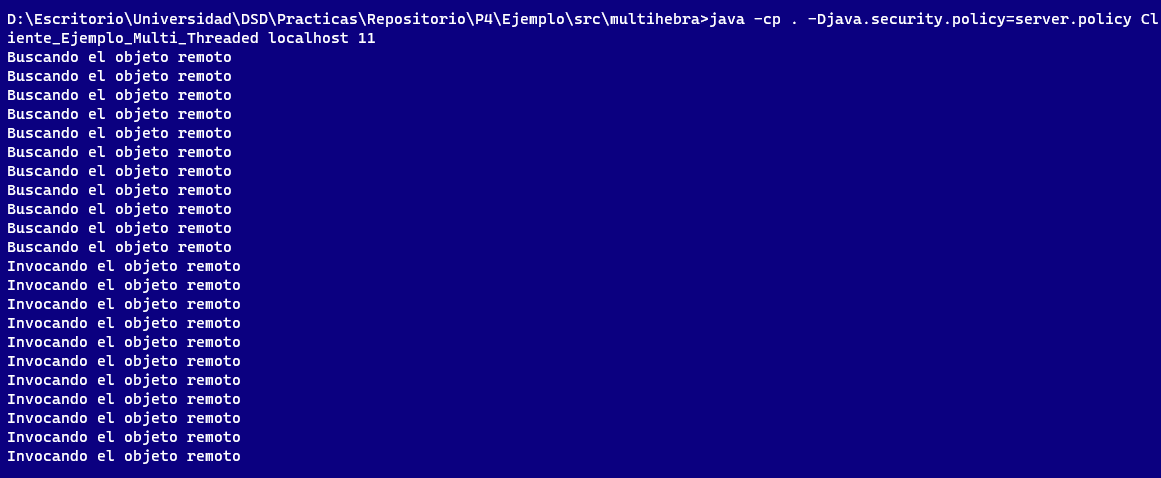
2º Ejecutar el servidor

> java -cp . -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Ejemplo

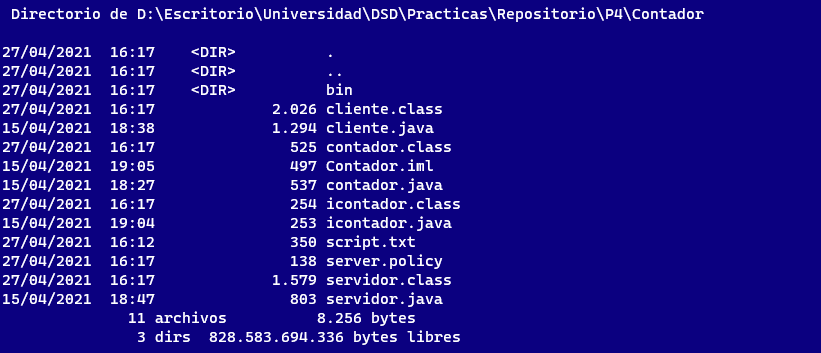


3º Ejecutar el cliente

> java -cp . -Djava.security.policy=server.policy Cliente\_Ejemplo\_Multi\_Threaded localhost 11

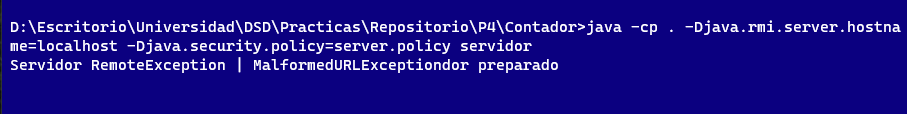


### Contador

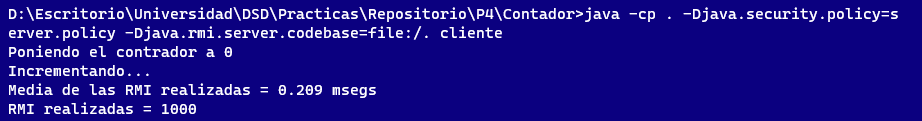


1º Ejecutar servidor

> java -cp . -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy servidor



2º Ejecutar cliente



> java -cp . -Djava.security.policy=server.policy -Djava.rmi.server.codebase=file:/. cliente

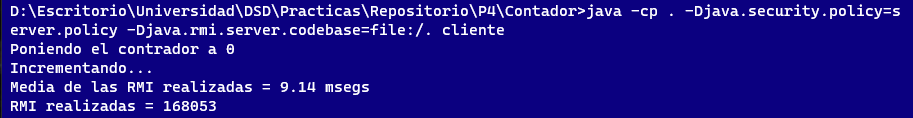
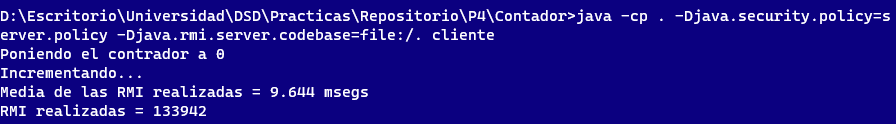
El programa cliente establece un valor 0 a una variable del servidor y comienza, en bucle, a incrementar su valor. Al final retorna el tiempo medio en ejecutar las peticiones.

Podemos fijarnos en varias cosas curiosas del ejemplo:

* No hemos necesitado hacer el comando rmiregistry. Esto es porque he añadido una instrucción en el servidor que hace que el programa lo cree.



* La variable del servidor es compartida por todos los clientes, por lo que, a pesar que la operación incrementar sea idempotente, pueda hacer que falle la suma porque al principio los clientes establecen el valor de la variable a 0. Aumentemos el tiempo del bucle y usemos dos clientes a ver qué pasa…



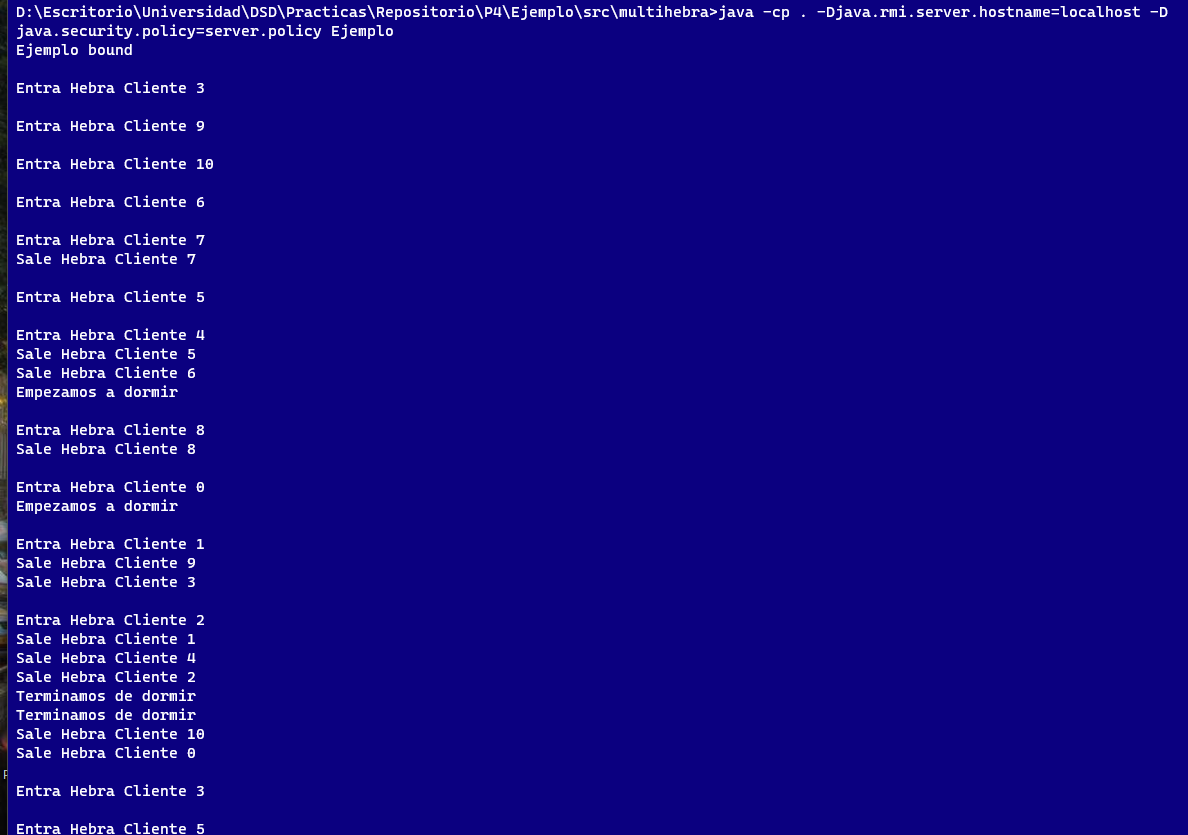
Nos dan resultados distintos. En nuestro ejercicio siguiente deberemos de tener en cuenta estas cuestiones.

## 

## 

## Ejercicio 1

**¿Qué ocurre con las hebras cuyo nombre acaba en 0? ¿Qué hacen las demás hebras? ¿Se entrelazan los mensajes?**



Como vemos en este ejemplo de ejecución, se entrelazan los mensajes.

Cuando una hebra tiene un identificador que acaba en 0, la hebra duerme durante 5 segundos.

if (mensaje.endsWith("0")) {

try{

System.out.println("Empezamos a dormir");

Thread.sleep(5000);

**Uso del modificador syncronized.**

Utilizando el modificador, en cambio, **no** se entrelazan los mensajes debido a que las hebras se ejecutan por orden en el servidor.

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

Como podemos ver en la documentación de java proporcionada por Oracle, los métodos que utilizan la palabra reservada “synchronized” tienen un comportamiento especial:

cuando una hebra ejecuta un método de este tipo en un objeto, todas las hebras que invocan el mismo método quedan suspendidas hasta que se termina la ejecución del método de la primera hebra que lo invocó.

# 

# Ejercicio 2 - Servidor replicado

## Introducción

Se pretende modelar un programa que siga la arquitectura cliente-servidor utilizando Java RMI. En este caso se compartirán varios objetos en el registro, que actuarán de servidores. Así pues, deberemos de encargarnos de que exista consistencia de datos entre los mismos pero conseguimos ganar en escalabilidad y tolerancia a fallos.

En concreto, se ha desarrollado un programa que puede trabajar con un número indefinido de réplicas del servidor. Además, cada réplica puede tener constancia de las réplicas que nos interesen. Para este ejemplo concreto, he creado tres réplicas y cada una de ellas puede comunicarse con las demás, debido a que son compartidas en el registro y cumplen una interfaz.

El programa del cliente podrá conectarse a cualquiera de estas réplicas, pero éstas se pondrán de acuerdo (consenso) para mantener consistencia entre los datos y garantizar que la carga esté balanceada.

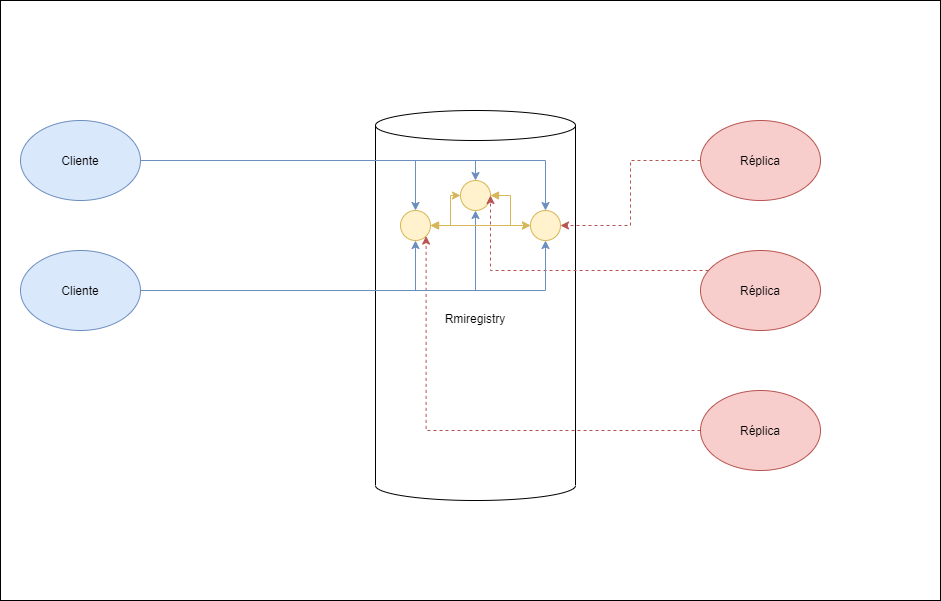
## Desarrollo

En este diagrama podemos ver un resumen completo de la arquitectura de nuestra aplicación. Los clientes se comunican con las distintas réplicas compartidas en el registro de RMI.

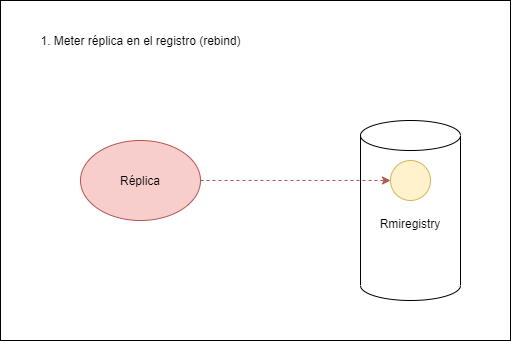
Observamos que las réplicas pueden comunicarse entre ellas, permitiendo así mantener la consistencia entre los datos y compartir información. Vemos además que esta comunicación se hace de forma transparente al cliente.

Debemos de tener en cuenta que cuando utilizamos la tecnología Java RMI trabajamos con interfaces en vez de con clases. Así pues, el cliente se comunica con la réplica a través de una interfaz, y una réplica se comunica con otra a través de otra interfaz (ambas son implementadas por la clase réplica).

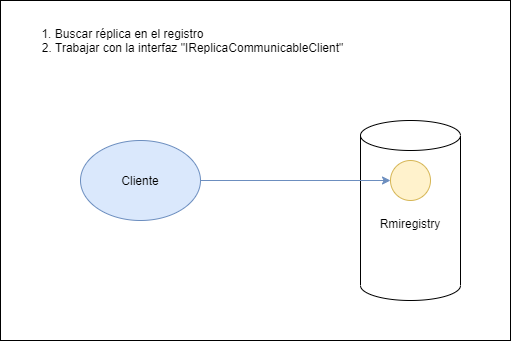
Así, conseguimos restringir las operaciones que puede hacer el cliente con las réplicas.

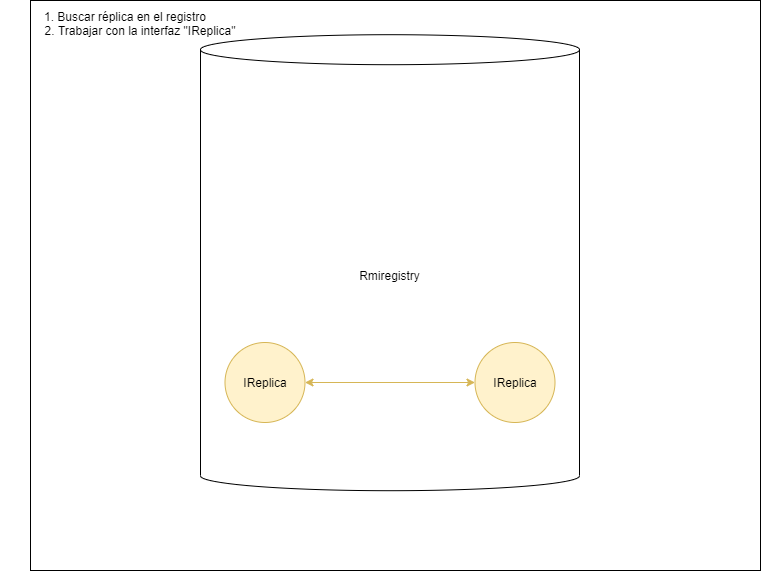


Para conseguir montar esta arquitectura, lo primero que se debe de crear es el registro. Después se   
deben de compartir todas las instancias de las réplicas que después serán visibles a los clientes.



Ahora que las réplicas se encuentran en el registro, los clientes pueden comunicarse con ellas buscando en el mismo. La comunicación con las mismas se llevará a cabo mediante una interfaz “IReplicaCommunicableClient”, que le ofrecerá los métodos necesarios para que el cliente se pueda comunicar con el servidor.



Ahora que el cliente puede establecer comunicación con las réplicas, éstas deben de poder comunicarse además para mantener las reglas de negocio. Por ejemplo, si un cliente pide registrarse en una réplica, deberá preguntar a las demás si ese cliente ha sido ya registrado. En caso afirmativo, se enviará un mensaje de error al cliente y en caso de que no haya sido registrado en ninguna, se registra en la que menos carga tenga. Por tanto debería de preguntar a las réplicas cuántos clientes tienen para saber dónde registrar al cliente. Esto se hace buscando la réplica en el registro. Se trabaja con una interfaz distinta, “IReplica”.

## Interfaces

### IReplicaCommunicableClient

Permite la comunicación de un cliente con la réplica

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | **donateCOMMClient**(java.lang.String email, java.lang.String password, float amount)  Añade una nueva donación |
| java.lang.Integer | **getClientNumDonationsCOMMClient**(java.lang.String email, java.lang.String password)  Devuelve el número de donaciones realizadas por un cliente |
| float | **getClientTotalAmountCOMMClient**(java.lang.String email, java.lang.String password)  Devuelve el total donado por un cliente |
| float | **getTotalAmountCOMMClient**(java.lang.String email, java.lang.String password)  Devuelve el total donado en el servidor |
| void | **registerCOMMClient**(java.lang.String email, java.lang.String password)  Registra un nuevo cliente |
| java.lang.String | **toStringCOMMClient**()  Muestra información del estado de la réplica |

### IReplica

Permite la comunicación entre réplicas

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| java.lang.Boolean | **checkCredentials**(java.lang.String email, java.lang.String password)  Comprueba si las credenciales son correctas. |
| java.lang.Boolean | **checkHasDonated**(java.lang.String email)  Comprueba si un cliente ha donado |
| void | **donate**(java.lang.String email, float amount)  Hace una nueva donación |
| **Client** | **getClient**(java.lang.String email)  Obtiene un cliente ya registrado |
| java.lang.Integer | **getNumClients**()  Obtiene el número de clientes en la réplica |
| java.lang.Integer | **getNumDonationsMadeByClient**(java.lang.String email)  Obtiene el número de donaciones hechas por un cliente |
| float | **getTotalAmountDonated**()  Obtiene el subtotal donado en la réplica |
| float | **getTotalAmountDonatedByClient**(java.lang.String email)  Obtiene el total donado por un cliente |
| java.lang.Boolean | **isClientRegistered**(java.lang.String email)  Comprueba si un cliente ha sido registrado |
| void | **register**(java.lang.String email, java.lang.String password)  Registra un nuevo cliente |

## Implementación

### Réplica

Clase que implementa las dos interfaces definidas arriba.

|  |  |
| --- | --- |
| **Modifier and Type** | **Field and Description** |
| private java.util.Map<java.lang.String,**Client**> | **clients**  Map de clientes en el que la clave es el identificador único de cada uno (email) |
| private java.lang.String | **HOST** |
| private java.lang.String | **identifier**  Identificador de la réplica en el registro |
| private java.util.ArrayList<java.lang.String> | **idOtherReplicas**  Listado de identificadores de otras réplicas en el registro |
| private java.util.ArrayList<**IReplica**> | **otherReplicas**  Réplicas que han sido obtenidas del registro. |
| private java.lang.Integer | **PORT** |
| private java.rmi.registry.Registry | **registry** |
| private float | **total**  Total donado en la réplica |

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | **addOtherReplica**(java.lang.String idOtherReplica)  Añade una réplica al listado de réplicas que puede ver. |
| private void | **incTotal**(float amount)  Incrementa el total donado en la réplica. El método tiene el modificador “syncronized” debido a que se debe de ejecutar en exclusión mutua para garantizar que el total siempre este actualizado correctamente. |
| IReplica | whereIsClient(java.lang.String email)  Devuelve la réplica en la que se encuentra el cliente registrado con el identificador email |

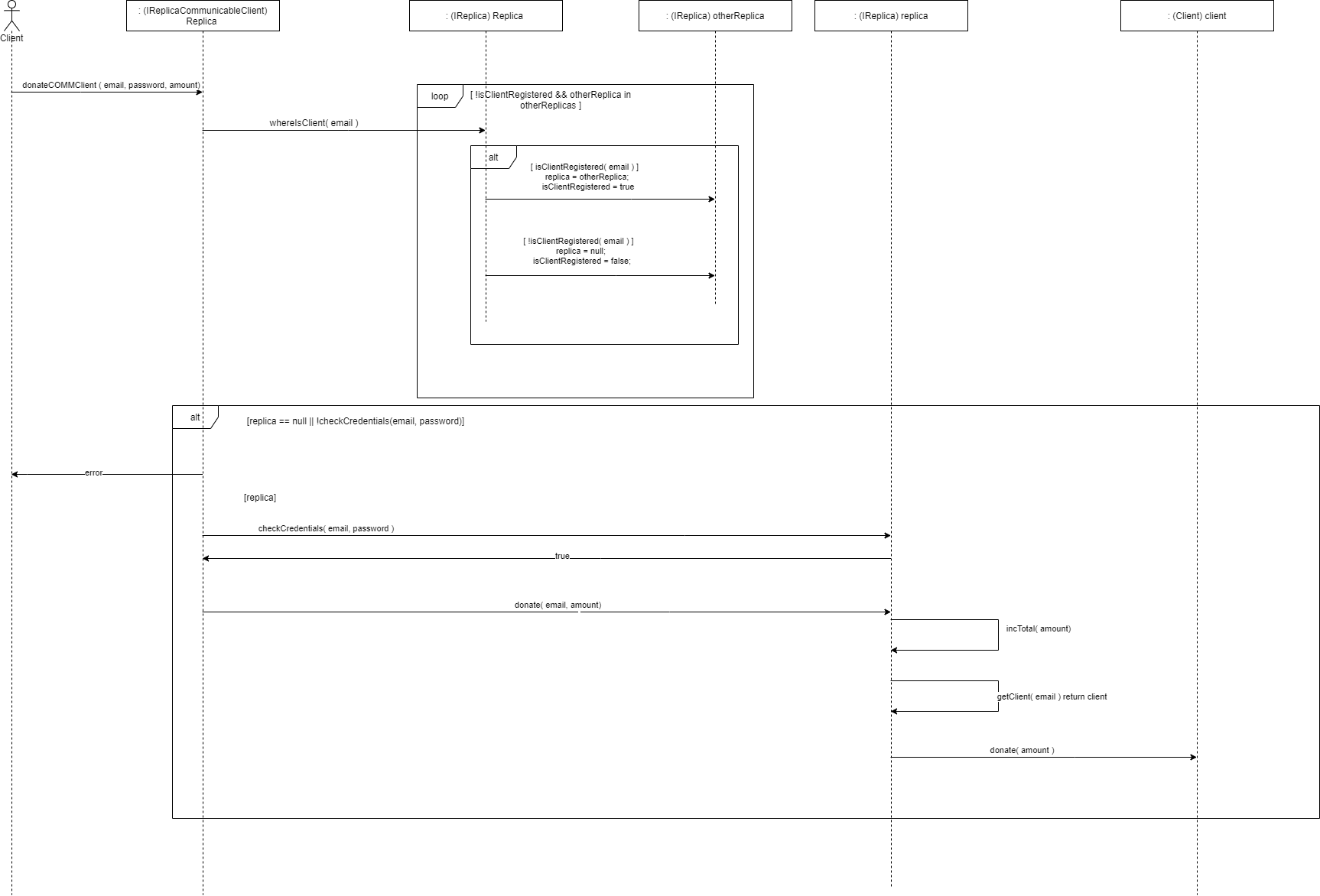
## Comunicación

Las réplicas pueden comunicarse con las demás debido a que cada una tiene un listado de las réplicas que hay en el registro. Así, cuando un cliente quiera realizar una operación con el servidor, la réplica comprobará si está registrado en alguna réplica con el método “whereIsClient”. Si el cliente está registrado se comprobará que cumple con las reglas del negocio, tales como que las credenciales sean correctas, el usuario esté registrado y/o que haya donado previamente.

1. Buscar cliente en el resto de réplicas
2. Comprobar las credenciales
3. (Opcional) Comprobar que ha donado
4. Realizar la operación correspondiente

Así pues, la estrategia seguida para la cooperación entre réplicas ha sido la de descubrimiento. Cada una de ellas guarda un subtotal de la cantidad donada y cuando se necesite conocer la cantidad total se buscarán las réplicas en el registro y se sumarán todos los subtotales.

Vemos a continuación un ejemplo de comunicación entre servidores con el caso de uso de donar.

Caso de uso donar.

## Ejecución

Ejecutamos el servidor

> java -cp . -Djava.rmi.server.codebase=file:/. -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy AppServer

Ejecutamos en otra terminal el cliente

> java -cp . -Djava.security.policy=server.policy -Djava.rmi.server.codebase=file:/. AppClient

Para mostrar la ejecución he incluido un vídeo en el que realizo todos los casos de uso con el sistema.

## Extras

Como extras de la práctica propuesta, se ha tenido en cuenta:

* No sólo se trabaja con dos réplicas. En este caso la implementación se ha realizado con tres réplicas, aunque se podrán añadir indefinidamente y el sistema funcionaría sin cambiar el código.
* El administrador puede establecer la visibilidad entre réplicas.
* Cada uno de los subtotales es accedido en exclusión mutua.
* Se ha creado un menú, con el que el cliente puede interactuar con el servidor. Primero se elige qué acción desea realizar y posterior a ello, se elige la réplica con la que quiere interactuar. Si, por reglas de negocio el usuario no puede interactuar con esa réplica, transparentemente a él esta réplica buscará la que necesita el usuario y será la encargada de realizar las operaciones.
* Se añaden operaciones:
  + El cliente puede ver el subtotal donado por él
  + El cliente puede ver la cantidad de donaciones realizadas por él
* Todas las operaciones son idempotentes y la comunicación no guarda el estado. Esto se hace para asemejar la comunicación al protocolo HTTP. El cliente en cada acción deberá de enviar la información necesaria para realizar la operación en el servidor.
* Se añaden operaciones extra a modo de middleware que aumentan la seguridad del sistema como la comprobación de credenciales.